Driving the Composition of Runtime Platforms by Architectural Knowledge*

Lothar Baum[1], Martin Becker[1], Lars Geyer[1], Georg Molter[1], Peter Sturm[2]

[1]System Software Research Group
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
{lbaum,mbecker,geyer,molter}@informatik.uni-kl.de

[2]System Software and Distributed Systems
University of Trier
D-54286 Trier, Germany
sturm@informatik.uni-trier.de

ABSTRACT

Reusing approved components is an attractive approach for the customization of runtime platforms in an economically sensible manner. However, the successful transition from particular requirements to a suitable architecture including appropriate components heavily relies on the expertise of the system designers. In this paper, we propose an architecture-driven approach to support runtime platform developers in the composition of customized platforms. Central to this approach is the explicit consideration of architectural aspects on an intermediate level of description. At this level, the appropriate matching of requirements against properties of available components is controlled by formalized architectural knowledge. With SDL patterns and design spaces we present two techniques for performing this mapping process.

INTRODUCTION

The development of today's large-scale distributed applications is highly dependent on the provision of adequately customized runtime platforms. The construction of these platforms requires a significant development effort. Component-based software technology can offer a noticeable improvement in this respect: reusing successfully deployed –possibly generic– components from earlier software projects promises to yield better, e.g. more robust, error-free, and reliable systems in less time. A clear indication is the trend towards distributed object busses [16], or towards component-based and object-oriented operating systems [5],[6],[8].

The main difficulties of component technology are finding the right components for reuse, and adapting the components to the envisioned system or vice versa. These problems are often only solved in an ad hoc manner. Component descriptions typically formalize functional properties only, making the search for right components heavily dependent on implicit knowledge and long-time experience. Furthermore, components are often viewed as separate entities, at best denoting interface constraints concerning needed companion components. Without a given component infrastructure or appropriate composition rules, successful system composition once again relies on the developer's experience.

The problem is further worsened by the fact that most non-functional requirements on a runtime platform cannot be met by individual components but only by the system as a whole. To this end, the system's structure has to be designed to address aspects such as security, reliability, concurrency control, multimedia support, and scalability. Concerning operating system services, particular demands for efficiency and reliability make such structural decisions especially delicate. The aspects of system structure –or in a more comprehensive sense: the system's architecture– therefore has to be seen as a key issue for the realization of effective and efficient runtime platforms. It can neither be handled implicit nor can it be treated separate from component selection, customization, and composition. The envisioned architecture of the runtime platform has to be the driving force in the development process.

As a consequence, components have to be viewed in their *architectural context*, i.e. the set of architectural decisions and assumptions under which they were built. Component descriptions have to put emphasis on formalizing non-functional and especially structural aspects of the component's intended environment. Such component descriptions can then be combined with techniques for capturing design experience. By these means, the mapping of architectural constraints to appropriate

*This research was supported by the Deutsche Forschungsgemeinschaft as part of the Special Research Project 501.

components can be supported and even partially automated by tools.

The next section will discuss the term architecture in more detail and will point out the importance of architecture for a system's non-functional properties. It is followed by a general presentation of our architecture-driven approach in section three and a more detailed discussion of two techniques to perform the mapping of architectural requirements to reusable components in section four. A look at further work and our plans for experiments quantifying the effect of an architecture-driven approach conclude the paper.

THE ROLE OF SOFTWARE ARCHITECTURE IN COMPONENT TECHNOLOGY

Especially in component-based software development it is tempting to view a software system as a set of individual and exchangeable components. In the case of common component infrastructures such as CORBA [16], this view might even seem to be correct to a certain extent. However, there are a lot of system properties that do not result from the presence or absence of specific components. Especially non-functional system properties are rather the effect of the overall system structure – or to speak in terms of object busses: depending on the component infrastructure itself.

In order to meet requirements such as fault-tolerance or high availability, e.g. the complete system has to be designed so that no module forms a single point of failure, that critical modules are replicated, and appropriate voting mechanisms assure correct results. For a runtime platform to meet tight real-time constraints, not only the right scheduling component has to be chosen, but other aspects such as the type of inter-process communication (IPC), the number of allowed processes, or appropriate buffer sizes have to be considered.

All the aspects mentioned are facets of the system's architecture. In this sense, software architecture can be characterized as an abstraction of various relevant properties of a software system [2]. An architecture comprises a set of views on different system characteristics such as structure, communication protocols, or component semantics. Each of the different views emphasizes specific aspects of the entire system, abstracting from details irrelevant for the particular purpose. For example, structural views omit the algorithmic design of the system. Conversely, descriptions of the semantics of subsystems or components abstract from structural aspects. All these views are set in relation to each other so that their interplay is made explicit.

An underlying and integral part of an architecture is the rationale behind the different architectural aspects. The rationale captures the motivation for the choice of style, elements, and form [18]. For example, the criteria for the decomposition of the overall functionality into subsystems are part of the architecture just as the decomposition itself is.

Furthermore, the architecture serves as a guideline for the assignment of non-functional system properties to specific components, as far as this is possible. In cases where such an assignment cannot sensibly be made, the architecture itself is the carrier of the respective non-functional property – as in the above examples of fault-tolerance or time constraints.

Consequently, software architecture is a pervasive aspect of software development. In current practice, however, it is unfortunately too often involved just implicitly, e.g. by using a specific type of black-box component infrastructure. As a result, important non-functional requirements have to be considered on an informal basis only, e.g. by exploiting the expertise of experienced developers. This approach is insufficient for various reasons [4],[6]. By making software architecture an explicit and central aspect of software development, design decisions can be traced more easily and the development process can be supported by appropriate tools.

We therefore propose an architecture-driven, component-based approach for the development of customized runtime platforms. It combines the benefits of component reuse with an explicit consideration of architectural aspects. The envisioned architecture of the runtime platform can serve as the central element for controlling component selection, customization, and composition.

INTEGRATION OF ARCHITECTURE-DRIVEN AND COMPOSITIONAL APPROACHES

In order to explicitly consider structural aspects, our approach starts with descriptions of the envisioned runtime platform, reflecting architectural requirements on the platform. This may concern the architectural style to be used, a decomposition strategy, the types of needed components, communication mechanisms, interfaces, or protocols. As for notation, known and preferably semi-formal or formal techniques such as different types of ADLs [10],[15] and IDLs [17] are used.

The architectural requirements are derived from the application's properties and the overall requirements on the platform. Relevant application properties in this context are, e.g., the application's degree of parallelism, the coupling between application components, their communication frequency and synchronization requirements, or time and memory constraints. Such properties typically influence the overall platform architecture and are not supported by single components.

The derivation of appropriate architectural requirements concerning the runtime platform is part of the conventional design process. As such it is basically a creative and manual activity. However, it can be supported, e.g., by checklists combined with known architectural patterns. In such a way, experience with similar projects can be used to direct the derivation of architectural requirements towards a set of approved runtime platform architectures. Such approved architectures will to a certain degree provide guaranties for meeting important non-functional requirements.

Complementary to this information about the required system structure, there is information associated with each of the available components describing its respective architectural context. Components in this sense are not developed stand-alone, but they are designed for a specific type of runtime platform or a set of somewhat similar types of platforms. A description of constraints concerning this environment a component was designed for is incorporated in the component description. This concerns, e.g., the component's structural embedding together with dependent components, interfaces and protocols, the component's semantics, and its role in the overall system. Favorably, the chosen description techniques should be compatible to those used for stating the architectural requirements. This allows for matching the descriptions without informal or implicit knowledge that would otherwise be necessary if notations have to be transformed manually.

Based on these two kinds of information, those components can be selected that fit well into the system under development with respect to its prospective architecture. Ideally, the envisioned overall system architecture resulting from the architectural requirements matches well against a known architecture. This dramatically eases the selection of components and especially their integration. In the more general case, the architectural properties of the runtime platform can be matched against the descriptions of the components in the reuse pool (cf. figure 1). This extends common practice, where this matching is performed on the basis of required functionality only, and the necessary experience which structures are favorable is not part of the model.

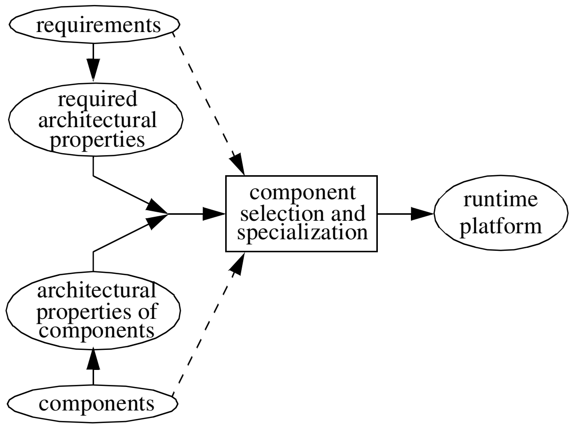

Fig. 1: Architecture-driven composition of runtime platforms. Dotted lines indicate the conventional flow of information, now extended by the architectural descriptions.

In the subsequent section, we present in more detail two techniques for an architecture-based mapping from requirements to components. The starting-point for both approaches is a set of runtime platform requirements obtained by an analysis of the application.

TECHNIQUES FOR CAPTURING ARCHITECTURAL KNOWLEDGE

The first approach is based on using SDL to state the design and thereby specific architectural facets of the envisioned runtime platform. SDL [12] is a widely accepted specification and design language and the technique of choice especially for describing structural and functional requirements. In our context, an SDL description is used as the intermediate step between architectural requirements and the specification of individual components (cf. figure 2). Along such a design of the runtime platform, required types of functionality such as a scheduler, a file system, and a memory manager can be identified. The design will furthermore exhibit the relationships between the modules, e.g. concerning communication protocols or interfaces needed.

Together with the component descriptions we provide a special type of patterns [9] to describe approved design solutions meeting specific requirements. The problem contexts of these so-called *SDL patterns* are given as SDL fragments. The solution part then provides a more detailed specification or even a working implementation. Since patterns in general promise the highest benefit if the environment they are applied in is pattern-aware, the runtime platforms have to be designed within the vocabulary of our patterns.

However, because of a common agreement on typical abstractions, functionality, and runtime platform designs, the deployment of patterns is a promising approach to reusing architectural solution elements. SDL patterns have already been applied successfully to communication protocols [11], and extending their deployment to the realm of runtime platform architectures has shown comparable effects.

The patterns are used to refine the design of the runtime platform: the coarse grained design resulting from the requirements analysis is scanned for elements matching the problem context of a pattern. The solutions provided by the patterns can generally be manifold: a more detailed system design expressed in an ADL, a set of concrete components, or a more detailed SDL specification for parts of the runtime platform. Typically, the patterns will identify a concrete set of components, thus bridging the gap between requirements and the reuse pool. As an example, consider the requirement of reliability for a file service. The SDL design of the runtime platform could reflect this requirement by specifying a file service block type that is instantiated n times. The matching SDL pattern would then provide a number of file service components together with a tailored voter component appropriately interacting with the replicated services. Decisions about the number of components or the type of the voter can, e.g., be based on specially qualified SDL names. For the time being, pattern matching is done manually, but semi-automatic support for matching SDL fragments is possible at least for a subset of our patterns.

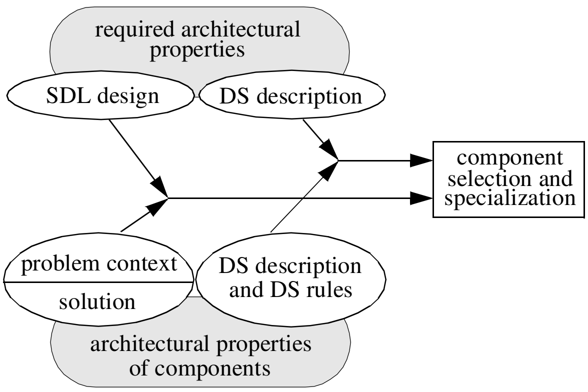

Fig. 2: SDL patterns and design spaces for matching requirements with component descriptions.

In our second approach, we deploy the concept of *design spaces* as originally presented in [13],[14] and extended in [3]. Design spaces provide a uniform, semi-formal way for describing and classifying both requirements to and properties of software artefacts. A design space is spanned by a set of *dimensions* which identify relevant criteria for characterizing systems. Those dimensions referring to requirements or properties concerning a software artefact's functionality span the so-called *requirements space* of the overall design space. The remaining dimensions describe architectural or structural properties; the sub-space spanned by these dimensions is therefore called the *structural space*. Dimensions are typically not independent from each other. To this end, relationships between the dimensions –so-called *correlations*– can be used to express favorable or unfavorable combinations of choices.

The demands concerning specific classes of services are stated by characterizations along the design spaces for those services. A scheduler could be characterized, e.g., along the dimensions of scheduling discipline, real-time capabilities, latencies, and maximum number of threads. In such a way, a design space not only provides a technique for capturing requirements, but also supplies a guideline for identifying important issues to be addressed during requirements specification and system design. Correlations within the requirements sub-space can then be used for checking consistency constraints in the requirements.

Just the way SDL is applied in our first approach, now the design space notation is used as a common vocabulary for requirements description and components specification alike. Design spaces can support the matching between requirements and components on several levels. Descriptions concerning functional requirements can be directly matched against the descriptions of available components. Again considering the example of the reliable file service, the design space of file services could have a dimension characterizing optimization criteria. A selection in this dimension might denote an optimization for small files. A comparison of the requirements sub-space profiles of requirements and available file service components then easily identifies those components optimized for small files. This matching can of course be extended to the structural sub-spaces allowing to compare non-functional and architectural requirements and properties. Correlations within the structural sub-space express favorable and unfavorable combinations of design decisions, thereby giving hints about which components can sensibly combined.

Furthermore, correlations between requirements dimensions and structural dimensions can be interpreted as design rules providing guidance for the mapping between requirements and structural design decisions. Applying these rules to a given requirements profile results in a design space characterization, which can be matched against the characterizations of the components. If the reliable file service, e.g., would be required to deploy caching techniques, design space correlations could state a favorable combination of this caching feature with a certain type of memory management. Applying such a design rule would result in additionally selecting an appropriate memory manager offering special support for a caching file service.

In order to support this matching process, we have developed a prototype tool to model design spaces and especially to give graphical support in the evaluation of design space correlations.

The two approaches of SDL patterns and design spaces focus on separate aspects of the system under development. The pattern approach addresses aspects easily expressed in SDL, namely structure and functionality. Especially non-functional aspects can in turn better be expressed with design spaces. Typically, distinct aspects will either be handled by patterns or design space descriptions. The approaches can therefore both be deployed extending on each other. It is also possible to extract a design space description from an existing SDL design if appropriate patterns are not available or cannot be applied sensibly. To support this, we have developed a tool for evaluating SDL specifications. It is able to determine certain key properties of the system under development such as the degree of parallelism, types of used IPC, or postulates of memory usage. This information may then be used as input for the system's design space classification, thus allowing to apply our second approach.

Conclusion and Further Work

Directly matching the required functionality against the descriptions of available components is neither easy nor sufficient. We therefore propose to match architectural information derived from the runtime platform requirements against the architectural context of available components. Introducing such an intermediate step bears important advantages. Traceability and repeatability of development steps are increased, because the critical and creative decisions are concentrated in few places. By using semi-formal description techniques, a significant part of this process can be supported by tools. Design decisions are explicitly documented, which facilitates their reuse as well as the understanding and maintenance of the developed system.

Our work is embedded in a special research project bringing together researchers from areas such as operating systems, software engineering, formal methods, and databases. Experience in this project shows that both SDL patterns and design spaces can already give substantial support when used manually. The development of the aforementioned tools has been completed and we are planning to perform further experiments to quantify their effect on the overall composition process. We are also heading towards a further refinement of the presented concepts –for example by extending the pool of patterns– and especially towards the integration of the concepts into a comprehensive process model. An interesting question in this context is, e.g., how the technique of evaluating SDL designs can also be applied to derive requirements to the runtime platform directly from early application designs.

Literature

[1] Allen, R.; Garlan, D.: *Formalizing Architectural Connection*, Proc. 16th. Int'l Conf. on Software Engineering, IEEE Computer Soc. Press, Los Alamitos, Calif., 1994

[2] Bass, L.; Clements, P.; Kazman, R.: *Software Architecture in Practice*, Addison-Wesley, 1997

[3] Baum, L.; Geyer, L.; Molter, G.; Rothkugel, S.; Sturm, P.: *Architecture-Centric Software Development Based On Extended Design Spaces*, Proc. 2nd ARES Workshop (Esprit IV 20477), February '98, Las Palmas

[4] Biggerstaff, T.: *The Library Scaling Problem and the Limits of Concrete Component Reuse*, IEEE Int'l Conference on Software Reuse, Nov. 1994

[5] Campbell, R.; Hine, J.; Russo, V.: *Choices for mission critical computing*, Studies in Computer and Communication Systems, IOS Press, Amsterdam, 1992

[6] Castelli, G.; Ragazzini, G.: *EOS: A Real-Time Operating System Adapts to Application Architectures*, IEEE Micro, Vol. 15, No. 5, Oct. 1995

[7] Doublait, S.: *Standard Reuse Practices: Many Myths vs. a Reality*, Standard View, ACM Perspectives on Standardization, Vol. 5, No.2, June 1997

[8] Ford, B.; Back, G.; Benson, G.; Lepreau, J.; Lin, A.; Shivers, O.: *The Flux OSKit: A Substrate for Kernel and Language Research*, Proc. of 16th ACM Symp. on Operating Systems Principles, Saint-Malo, Oct. 1997

[9] Gamma, E. et. al.: *Design Pattern: Elements of Reusable Object-Oriented Design*, Addison-Wesley, Reading, Mass., 1994

[10] Garlan, D.; Monroe, R.T.; Wile, D.: ACME: *An Architecture Description and Interchange Language*, Technical Report, Carnegie Mellon Univ., Pittsburgh, 1996

[11] Geppert, B.; Gotzhein, R.; Rößler, F.: *Configuring Communication Protocols Using SDL Patterns*, Proc. of the 8th SDL Forum, SDL`97, Paris/Evry, France, Sep. 1997

[12] ITU-T: *CCITT Specification and Description Language* (SDL), Z.100, 1994

[13] Lane, T.G.: *Guidance for User-Interface Architectures*, in: Garlan, D.; Shaw, M.: Software Architecture – Perspectives on an Emerging Discipline, Prentice Hall, New Jersey, 1996

[14] Lane, T.G.: *Studying Software Architecture Through Design Spaces and Rules*, Technical Report CMU/SEI-90-TR-18, Carnegie Mellon Univ., 1990

[15] Luckham, D.C. et al: *Specification and Analysis of System Architecture using Rapide*, IEEE Trans. on Software Engineering, Apr. 1995

[16] Object Management Group: *The Common Object Request Brocker: Architecture and Specification (CORBA)*, OMG / Framingham, M.A., Rev. 1.3, Dec. 1993

[17] Object Management Group: *OMG IDL Syntax and Semantics*, OMG TC Document 97-05-03, May 1997

[18] Perry, D.E.; Wolf, A.L.: *Foundations for the Study of Software Architecture*, ACM SIGSOFT Software Engineering Notes, Vol. 17 No. 4, Oct. 1992